Variations sur le style architectural Pipe and Filter

Songsakdi Rongviriyapanish, Nicole Lévy

Variations sur le style architectural *Pipe & Filter*

S. Rongviriyapanish[1] and N. Lévy[1,2]

[1] LORIA
BP. 239, F-54506 Vandœuvre-lès-Nancy, France
rongviri@loria.fr
[2] PRISM, Université de Versailles St-Quentin
10, 12 Av. de l'Europe, 78140 Velizy, FRANCE
Nicole.Levy@prism.uvsq.fr

Abstract. La formalisation des styles architecturaux permet une définition précise de la notion de variation ainsi que le contrôle du fait qu'une spécification soit conforme au style choisi.
Dans cet article, nous proposons une formalisation du style *Pipe & Filter* ainsi que plusieurs variations sous forme des schémas paramétrés. Le langage de spécification formel LOTOS est utilisé comme langage de description d'architectures. Nous utilisons ce style et ses variations pour développer une étude de cas d'un produit de convolution. Différentes solutions équivalentes comportementalement sont étudiées et comparées à l'aide de l'environnement CADP.

Mots Clés

Développement formel de spécifications, styles architecturaux et variations, architecture de logiciels, LOTOS.

1 Introduction

La spécification de l'architecture d'un logiciel est une étape bien identifiée dans le cycle de vie. Il s'agit de décrire un système en termes de composants et de connecteurs [AAG93] de telle manière que la spécification décrive le comportement désiré du système. Les styles architecturaux sont des techniques facilitant l'identification et la définition des composants et des connecteurs [Me98].

Apparaît alors le problème de la représentation des styles pour fournir d'une part des critères de décision concernant la conformité d'une spécification à un certain style, et d'autre part des aides au développement de spécifications d'architectures.

Nous proposons de décrire un style d'architecture à l'aide de schémas paramétrés de spécifications. Ces schémas sont des morceaux de spécifications dans lesquels certaines parties sont remplacées par des variables typées et contraintes. Ils décrivent le comportement des composants, connecteurs ainsi que la configuration globale. L'instanciation de ces schémas se fait en donnant une valeur aux variables, cette valeur devant vérifier les contraintes. Ainsi, les variables sont les paramètres du style. Remarquons qu'une même variable peut être utilisée à plusieurs endroits d'un schéma ou dans plusieurs schémas d'un style. L'instanciation est automatisable.

Les avantages de notre approche sont les suivants :

- un schéma est une description formelle et n'est pas ambiguë;
- les spécifieurs ont à leur disposition des critères d'applicabilité d'un style;
- l'instanciation des schémas est facilitée par l'explicitation des paramètres;

- le développement d'une architecture se fait simplement par instanciation et composition des schémas;
- la conformité d'une spécification à un style peut être vérifié automatiquement;
- le développement des outils est facilité [EGY97].

Des langages formels sont proposés pour décrire des architectures de logiciels. De nouveaux langages ont été développés, mais ils sont encore dans une phase de maturation et peu sont outillés [Cle96].

Au lieu de définir un nouvel ADL (Architectural Description Language) [SG96], nous utilisons le langage de spécification formel LOTOS [BB87]. LOTOS est un langage adéquat pour décrire des styles architecturaux, en particulier pour établir une sémantique formelle de la communication entre composants. Plusieurs environnements de développement pour LOTOS existent. Nous utilisons CADP (Caesar/Aldebaran Distribution Package) [FGM+92], qui permet d'analyser et d'animer des spécifications LOTOS. Remarquons que LOTOS a été normalisé par l'ISO [LOT87].

Dans cet article, nous expérimentons une approche de développement d'architectures guidée par les styles. Notre approche est illustrée par la formalisation du style *Pipe & Filter*. Ce style et ses variations sont utilisés pour développer différentes spécifications équivalentes d'un produit de convolution. Ces spécifications sont analysées et comparées à l'aide de l'outil CADP.

Dans la section 2, l'approche générale suivie pour formaliser des styles architecturaux est présentée. Dans la section 3, nous définissons le style architectural *Pipe & Filter*. Ce style est utilisé dans la section 4 pour développer la spécification d'un produit de convolution avec différentes variantes. Un résumé du langage LOTOS utilisé dans cet article est présenté en annexe.

2 Formalisation des Styles Architecturaux

2.1 Qu'est-ce qu'un style ?

D'après [NR97], la définition d'un style est composée de quatre parties : les composants et les connecteurs, la configuration globale, les contraintes architecturales et des variantes de composants et de connecteurs. Nous rajoutons une partie décrivant les caractéristiques du style. Ainsi, nous décrivons les styles de la manière suivante :

1. Composants et Connecteurs : les composants sont des unités de calcul ou de données. Leur interface est un ensemble de points d'interaction avec les connecteurs ou avec l'environnement. Les connecteurs sont utilisés pour modéliser les interactions entre les composants. L'interface d'un connecteur est l'ensemble des points d'interaction avec les composants auquel il est rattaché. Composants et connecteurs sont décrits à l'aide de schémas paramétrés.
2. Configuration globale : elle décrit la structure architecturale caractéristique du style, c'est à dire la manière dont doivent interagir les composants et les connecteurs. Elle est définie par un schéma décrivant le comportement général d'une spécification, la composition des composants et des connecteurs.
3. Caractéristiques : l'ensemble des paramètres du style, c'est à dire, l'ensemble des informations nécessaires à la définition de la spécification par instanciation des schémas. Ces informations sont dénotées par des variables typées et contraintes qui paramétrisent les différents schémas.

4. Contraintes architecturales : propriétés qu'une spécification doit vérifier afin d'être conforme au style, et ce, indépendamment du développement suivi, de l'utilisation ou non des schémas proposés. Ce sont des contraintes structurelles concernant la configuration globale ou comportementales concernant les composants et les connecteurs.
5. Variations : schémas décrivant des variantes de définition des composants et des connecteurs. Ces schémas décrivent des comportements équivalents comportementalement à ceux décrits initialement.

2.2 Styles architecturaux en LOTOS

Dans cet article, nous utilisons le langage de spécification formel LOTOS [BB87] et son environnement CADP (Caesar/Aldebaran Distribution Package) [FGM+92]. LOTOS est composé de deux parties :

- une algèbre de processus inspirée de celle de CCS [Mil80] et de CSP [Hoa85]. Sa sémantique est définie en terme de systèmes de transitions étiquetées;
- ACT-One [EM85] pour la description des structures de données à l'aide de types abstraits algébriques. Les types de données sont utilisés pour décrire les paramètres des processus ainsi que les valeurs qu'ils échangent.

Un modèle de conception d'un système exprimé en LOTOS se compose de deux parties : le comportement global du système, dénotée ci-dessous par la variable ***beh***, la définition des processus et des types abstraits algébriques utilisés dans ***beh*** dénotés ci-dessous par ***local_def***. Il faut de plus préciser la liste ***Ext_Gate_list*** des portes externes du système (portes de communication avec l'environnement) et la fonctionnalité ***func*** qui indique si le comportement de la spécification se termine (***func***= `exit`) ou non (***func***= `noexit`). La structure syntaxique est la suivante :

```
specification spec_name [Ext_Gate_list] :func
    behavior
          beh
    where
          local_def
endspec
```

Les idées de base de notre formalisation sont les suivantes :

- les composants et les connecteurs d'une architecture sont modélisés par des processus LOTOS et caractérisés par des schémas de spécifications contenant des variables à instancier;
- la configuration globale d'un style est définie par un schéma de communication LOTOS contenant elle aussi des variables. Ce schéma décrit la composition des composants et des connecteurs;
- les caractéristiques du style sont la liste des variables utilisées dans les schémas. Les variables représentent des parties de la spécification. Dans la suite, les variables seront écrites en italiques;
- pour obtenir une architecture concrète, le spécifieur doit choisir pour chaque composant et connecteur à définir, un schéma à instancier.

L'utilisation de schémas et de variables contraintes présente l'avantage de fournir des aides méthodologiques pour développer un système, tout en garantissant qu'il sera conforme par construction au style donné.

3 Formalisation du style *Pipe & Filter*

Les caractéristiques du style *Pipe & Filter* sont les suivantes [GS93] :

> Dans une spécification de style *Pipe & Filter* chaque composant a un ensemble d'entrées et un ensemble de sorties. Un composant lit des flots de données sur ses entrées et produit des flots de données sur ses sorties, [...] Les composants s'appellent des filtres. Les connecteurs de ce modèle servent de conduits aux flots, transmettant les sorties d'un filtre aux entrées d'autres filtres. Les connecteurs s'appellent des *pipes*[1]. [...] Les filtres doivent être des entités indépendantes : en particulier, ils ne doivent pas partager d'états avec d'autres filtres.

La figure 1 montre un exemple d'architecture de style *Pipe & Filter*. Un filtre peut avoir plusieurs pipes entrants et sortants. Les cycles sont autorisés.

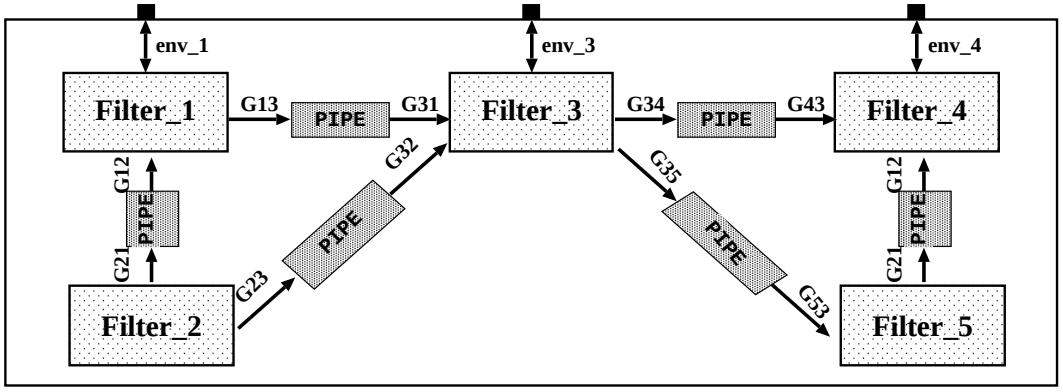

Fig. 1. A Pipe & Filter Architecture

3.1 Composants et connecteurs

Dans le style *Pipe & Filter*, tous les composants sont des *Filtres* et tous les connecteurs des *Pipes*.

Filtres. Un filtre réalise une transformation locale et incrémentale des données reçues sur ses portes d'entrée et envoie les résultats calculés sur ses portes de sortie. Ces résultats sont envoyés à d'autres filtres par l'intermédiaire des pipes ou à l'environnement. Un filtre est caractérisé par les variables suivantes :

- les noms tous différents de ses portes d'entrées $gate\_list\_IN$ et de sorties $gate\_list\_OUT$:
  $gate\_list\_IN = gate\_IN\_1, ... gate\_IN\_n \quad \wedge \quad \forall i,j \in 1..n \bullet gate\_IN\_i \neq gate\_IN\_j$
  $gate\_list\_OUT = gate\_OUT\_1, ... gate\_OUT\_m \wedge \forall i,j \in 1..m \bullet gate\_OUT\_i \neq gate\_OUT\_j$
- pour chaque porte, les types des valeurs reçues $T\_in\_1, \ldots T\_in\_n$ et celui de celles envoyées $T\_out\_1, \ldots T\_out\_m$
- éventuellement le type de données T de son état local

[1] Nous ne traduisons pas le terme *pipe*. En effet, la traduction par le mot "tube" nous paraît moins parlante.

– le calcul réalisé par le filtre, représenté par une fonction F dont la signature est la suivante :

$$F:\ T,\ T\_in\_1,\ \ldots T\_in\_n \rightarrow <\ T,\ T\_out\_1,\ \ldots T\_out\_m\ >$$

– les prédicats $pred\_j_{j \in 1..m}$ utilisés en tant que garde pour l'émission des résultats.

Le comportement d'un filtre peut être décomposé en trois comportements successifs :

– Une réception de n données sur les n portes d'entrée. Ces réceptions sont indépendantes et sont réalisées en parallèle :

```
data_reception_behaviour =
            gate_IN_1 ? v1 : T_in_1; exit(v1, any T_in_2, ... any T_in_n)
       ||| ...
       ||| gate_IN_n ? vn : T_in_n; exit(any T_in_1, ... any T_in_n-1, vn)
```

– Un calcul réalisé par appel de l'opération F. Le comportement décrit simplement la transmission des résultats calculés au comportement suivant :

```
computation_behaviour =
            exit(F(val, v1, ... vn))
```

– Une émission gardée par les prédicats $pred\_i$ des résultats sur les m portes de sortie. Ces émissions sont indépendantes et sont réalisées en parallèle :

```
result_transmission_behaviour =
            [pred_1] -> ( gate_OUT_1 ! w1 ; exit )
       ||| ...
            [pred_m] -> ( gate_OUT_m ! wm ; exit )
```

Le comportement d'un filtre peut alors être décrit par le schéma de processus suivant où les trois comportements décrits ci-dessus sont composés séquentiellement avec l'opérateur $\gg$:

Blocking Filter

```
process FILTER [gate_list_IN, gate_list_OUT] (val: T) : noexit :=
          Reception [gate_list_IN]
    >> accept v1: T_in_1,... vn: T_in_n in
          (Computation( val, v1, ... vn )
    >> accept val': T, w1: T_out_1, ... wm: T_out_m in
          (Transmission [gate_list_OUT]( w1, ... wm )

    >>     FILTER [gate_list_IN, gate_list_OUT] (val'))
where
    process Reception [gate_list_IN] : exit( val: T, v1: T_in_1,... vn: T_in_n ) :=
        data_reception_behaviour
    endproc
    process Computation( val: T, v1: T_in_1,... vn: T_in_n ) :
                        exit ( val': T, w1: T_out_1, ... wm: T_out_m  ) :=
        computation_behaviour
    endproc
    process Transmission [gate_list_OUT] ( w1: T_out_1,... wm: T_out_m):
                        exit :=
        result_transmission_behaviour
    endproc
endproc
```

Pipes. Un pipe a deux portes : une porte d'entrée où il reçoit un flot de données et une porte de sortie où il le renvoie, sans aucune transformation et dans le même ordre. Un pipe

se caractérise par le nom de ses uniques portes d'entrée et de sortie, *gate_IN* et *gate_OUT* et le type de données *V* des valeurs reçues et renvoyées. Son comportement peut être décrit par le schéma de processus suivant :

```
process PIPE [gate_IN, gate_OUT]: noexit :=
   gate_IN ? x : V;
   gate_OUT ! x ;
   PIPE [gate_IN, gate_OUT]
endproc
```

3.2 Configuration globale

La configuration globale d'une architecture de style *Pipe & Filter* décrit le comportement général d'une spécification, la manière dont interagissent les composants et les connecteurs. Elle peut être décrite par le schéma de comportement suivant :

```
beh =
       hide all_gates_IN, all_gates_OUT in
         ((     FILTER_1 [gate_list_IN_1, gate_list_OUT_1] (val_1: T_1)
            ||| ...
            ||| FILTER_p [gate_list_IN_p, gate_list_OUT_p] (val_p: T_p))
         |[ all_gates_IN, all_gates_OUT ]|
          (     PIPE_1 [gate_IN_1, gate_OUT_1]
            ||| ...
            ||| PIPE_q [gate_IN_q, gate_OUT_q]))
```

où

all_gates_IN, all_gates_OUT désignent l'ensemble des portes de communication entre les filtres et les pipes :

$all\_gates\_IN = \cup_{i=1}^{q} gate\_IN\_i$

$all\_gates\_OUT = \cup_{i=1}^{q} gate\_OUT\_i$

3.3 Caractéristiques

Un système de style *Pipe & Filter* est caractérisé par les attributs suivants :

- la liste des p filtres $(FILTER\_i)_{i \in 1..p}$ et pour chaque filtre, ses portes $gate\_list\_IN\_i$ et $gate\_list\_OUT\_i$;
- pour chaque filtre $FILTER\_i$, ses caractéristiques définies en section 3.1;
- la liste des q pipes $(PIPE\_j)_{j \in 1..q}$ et pour chaque pipe, ses portes $gate\_IN\_j$ et $gate\_OUT\_j$;
- la liste des portes externes du système (portes de communication avec l'environnement) Ext_Gate_list.

La configuration doit satisfaire les contraintes suivantes :

1. chaque porte d'entrée d'un pipe est une porte de sortie d'un filtre, et chaque porte de sortie d'un pipe est une porte d'entrée d'un filtre :
   $\forall j \in 1..q \ \exists! \ i \in 1..p \bullet gate\_IN\_j \in gate\_list\_OUT\_i$
   $\forall j \in 1..q \ \exists! \ i \in 1..p \bullet gate\_OUT\_j \in gate\_list\_IN\_i$

2. Le système communique avec son environnement par l'intermédiaire des filtres (les pipes ne communiquent pas avec l'environnement). Donc, la liste des portes externes doit être égale à l'union des portes des filtres n'étant pas une porte de pipe :
$$\mathit{Ext\_Gate\_list} = (\cup_{i=1}^{p}\ \mathit{gate\_list\_IN\_i}\ \cup\ \cup_{i=1}^{p}\ \mathit{gate\_list\_OUT\_i}) \setminus (\cup_{j=1}^{q}\ \mathit{gate\_IN\_j}\ \cup\ \cup_{j=1}^{q}\ \mathit{gate\_OUT\_j})$$
On remarque que ce type de contrainte permettrait de faciliter le développement, par exemple en déduisant la liste des portes externes du système ***Ext_Gate_list*** sans la demander explicitement au développeur.
3. Les filtres ne communiquent pas directement, ils n'ont pas de portes en commun :
$\forall i, j \in 1..p \bullet i \neq j \Rightarrow$
$(gate\_list\_IN\_i \cup gate\_list\_OUT\_i) \cap (gate\_list\_IN\_j \cup gate\_list\_OUT\_j) = \varnothing$
4. Les pipes ne communiquent pas directement, ils n'ont pas de portes en commun :
$\forall i, j \in 1..q \bullet i \neq j \Rightarrow \{gate\_IN\_i, gate\_OUT\_i\} \cap \{gate\_IN\_j, gate\_OUT\_j\} = \varnothing$

3.4 Contraintes Architecturales

Les contraintes architecturales décrivent des propriétés à satisfaire par un système pour être conforme au style *Pipe & Filter*. Elles sont décrites par des formules de la logique du premier ordre. Le système considéré est décrit par une spécification qui peut être obtenue sans utiliser les schémas donnés ci-dessus. Mais les propriétés à vérifier sont équivalentes à celles qui contraignent les variables caractéristiques du style.

Le système est dénoté par deux variables : ***beh*** représentant le comportement global et ***local_def*** représentant la définition des processus utilisés dans ***beh***. Les contraintes concernent la spécification entière ou seulement une partie. Nous supposons que les fonctions suivantes sont définies [2].

Ext_Gates :	*BEHAVIOR*	$\rightarrow$	*GATE_LIST*	Liste des portes externes (non cachées) d'une expression de comportement
Proc_Calls :	*BEHAVIOR*	$\rightarrow$	*PROC_CALL_LIST*	Liste des expressions d'appel de processus dans un comportement
Name :	*PROC_CALL*	$\rightarrow$	*IDENT*	Nom du processus appelé dans une expression d'appel de processus
Gates :	*PROC_CALL*	$\rightarrow$	*GATE_LIST*	Portes mentionnées dans une expression d'appel de processus
Proc_Def :	*LOCAL_DEF*, *IDENT*	$\rightarrow$	*PROC_DEF*	Rend la définition d'un processus d'une liste de déclarations locales

Les types (*BEHAVIOR*, *GATE_LIST*, *PROC_CALL_LIST*, ...) sont ceux de parties d'une spécification. Ce sont ceux des noeuds de l'arbre de la syntaxe abstraite représentant la spécification. De plus, nous supposons qu'un attribut ***Archi*** est associé à chaque définition de processus. Cet attribut permet de connaître le style ou le type de composant ou de connecteur d'un style d'un processus. Par exemple, une définition de processus dénotée par la variable ***f*** représentant un filtre aura `Filter` comme valeur de l'attribut ***Archi***, ce qui est noté : ***Archi(f)*** `= Filter`. Enfin et pour simplifier la notation, nous définissons les ensembles ***filters*** et ***pipes*** représentant les ensembles des filtres et des pipes d'un système comme suit :

$$filters(beh, local\_def) = \{f \in Proc\_Calls(beh) \mid Archi(Proc\_Def(local\_def, Name(f))) = \texttt{Filter}\}$$

[2] par exemple sur l'arbre de la syntaxe abstraite de la spécification.

$$pipes(beh, local\_def) = \{p \in Proc\_Calls(beh) \mid Archi(Proc\_Def(local\_def, Name(p))) = \texttt{Pipe}\}$$

Les contraintes sont soit structurelles (1-4) soit comportementales (5-6).

Contraintes :

1. Le système communique avec son environnement par l'intermédiaire des filtres, les pipes ne communiquent pas avec l'environnement :
   $\forall p \in pipes(beh, local\_def) \bullet Gates(p) \cap Ext\_Gates(beh) = \varnothing$
2. Les données sont transmises entre les filtres par l'intermédiaire des pipes (les filtres n'ont pas de portes en commun) :
   $\forall f1, f2 \in filters(beh, local\_def) \bullet Gates(f1) \cap Gates(f2) = \varnothing$
3. Chaque pipe a deux portes. Il relie une porte de sortie d'un filtre à une porte d'entrée d'un autre filtre :
   $\forall p \in pipes(beh, local\_def)\, \exists\, g_{in}, g_{out} \bullet Gates(p) = (g_{in}, g_{out}) \ \wedge$
   $(\exists f_{in}, f_{out} \in filters(beh, local\_def) \bullet g_{in} \in Gates(f_{in}) \ \wedge \ g_{out} \in Gates(f_{out}) \wedge f_{in} \neq f_{out})$
4. Tous les pipes sont indépendants (ils ne partagent pas de portes).
   $\forall p1, p2 \in pipes(beh, local\_def) \bullet Gates(p1) \cap Gates(p2) = \varnothing$
5. Tous les filtres doivent être conformes au comportement détaillé dans la section 3.1, c'est à dire qu'ils doivent être équivalents selon une relation d'équivalence comportementale[3] au schéma instancié définissant le comportement des filtres.
6. Tous les pipes doivent être conformes au comportement détaillé dans la section 3.1, c'est à dire qu'ils doivent être équivalents selon une relation d'équivalence comportementale au schéma instancié définissant le comportement des pipes.

3.5 Variations sur le style *Pipe & Filter*

Les variations représentent des solutions spécifiques de spécification des filtres et des pipes. Elles sont comportementalement équivalentes (selon l'équivalence de sûreté) aux schémas donnés dans la section 3.1.

Variations sur les Filtres. Comme vu dans le schéma général, le comportement d'un filtre est décomposé en trois comportements : la réception des données, le calcul et la transmission des résultats. Ces comportements peuvent s'exécuter séquentiellement comme dans le schéma général, ou en parallèle : par exemple, le filtre peut commencer à recevoir de nouvelles entrées avant d'avoir terminé son calcul. Nous appellerons ce comportement d'un filtre non bloquant (**non-blocking filter**). Dans ce cas, les trois comportements se composent en parallèle et se synchronisent sur deux portes internes cachées, *int_gate1* et *int_gate2*. Le schéma général de filtre donné dans la section 3.1 correspond au comportement de filtre bloquant (**blocking filter**). Le schéma de filtre non bloquant est décrit ci-dessous. Les trois variables *data_reception_behaviour*, *computation_behaviour* et *result_transmission_behaviour* dénotent les comportements définis dans la section 3.1.

[3] telle que l'équivalence de sûreté ou *safety equivalence* [FM91]

Non Blocking Filter

```
process FILTER [gate_list_IN, gate_list_OUT] (val: T) : noexit :=
    hide int_gate1,int_gate2 in
      ( Reception [gate_list_IN, int_gate1]
          |[int_gate1]|
        Computation [int_gate1, int_gate2]( val )
          |[int_gate2]|
        Transmission [gate_list_OUT,int_gate2] )
where
    process Reception [gate_list_IN, int_gate1] : noexit :=
           data_reception_behaviour
           >> accept v1: T_in_1,..., vn: T_in_n in
             ( int_gate1 !v1 ... !vn ;
               Reception [gate_list_IN, int_gate1])
    endproc
    process Computation [int_gate1,int_gate2] ( val: T ) : noexit :=
          int_gate1 ?v1:T_in_1... ?vn: T_in_n;
          computation_behaviour
        >> accept val': T, w1:T_out_1,..., wm:T_out_min
          ( int_gate2 !w1 ... !wm ;
            Computation [int_gate1,int_gate2] ( val') )
    endproc
    process Transmission [gate_list_OUT,int_gate2]: noexit :=
         int_gate2 ?w1:T_out_1 , ... ?wm:T_out_m ;
           result_transmission_behaviour
        >> Transmission [gate_list_OUT,int_gate2]
    endproc
endproc
```

Variations sur les Pipes. Le schéma général de pipe décrit le comportement souhaité: recevoir un message sur sa porte d'entrée et le renvoyer sur sa porte de sortie. Il est possible de découpler ces deux comportements, pour permettre par exemple plusieurs réceptions successives avant un envoi. Pour cela, un pipe doit avoir une mémoire tampon locale pour enregistrer les messages reçus en attente d'envoi. Cette mémoire tampon peut être bornée ou non. Pour modéliser de tels pipes, nous définissons les types `BUF` et `BOUNDED_BUF` munis des opérations `init`, `put`, `head`, `tail`, `empty`. Le type `BOUNDED_BUF` est de plus muni de l'opération `full`. Faute de place, nous ne décrivons pas ici ces types de données. Ils sont donnés dans le rapport interne [RL99]. Nous obtenons les deux schémas suivants de pipes:

Bounded Pipe

```
process PIPE [gate_IN, gate_OUT] ( A_Bounded: BOUNDED_BUF ) : noexit:=
    [not (full( A_Bounded ))] ->
      gate_IN ? x: T ;
      PIPE [gate_IN, gate_OUT] (put( x, A_Bounded ))
  []
    [not (empty( A_Bounded ))] ->
      gate_OUT ! head( A_Bounded ) ;
      PIPE [gate_IN, gate_OUT] (tail( A_Bounded ))
endproc
```

Unbounded Pipe

```
process PIPE [gate_IN, gate_OUT] ( An_Unbounded: BUF ) : noexit :=
      gate_IN ? x: T ;
      PIPE [gate_IN, gate_OUT] (put( x, An_Unbounded ))
  []
    [not (empty( An_Unbounded ))] ->
      gate_OUT ! head( An_Unbounded ) ;
      PIPE [gate_IN, gate_OUT] (tail( An_Unbounded ))
endproc
```

4 Étude de cas : un produit de convolution

Un produit de convolution définit une suite de valeurs Y_i en additionnant le produit de n valeurs constantes $W_1, \cdots, W_n$ par une suite de valeurs X_i.

$$Y_i = \sum_{1 \leq j \leq n} (W_j * X_{i+j-1}) \tag{1}$$

Un réseau systolique [Kun82,Gar89] avec n cellules identiques reliées exécutant le même algorithme peut être utilisé pour résoudre ce problème. La $j^{ième}$ cellule calcule à la $i^{ième}$ étape :

$$Y_j = (W_j * X_{i+j-1}) + Y_{j-1} \tag{2}$$

Ainsi, la $j^{ième}$ cellule reçoit la valeur Y_{j-1} et transmet Y_j à la cellule suivante. Les cellules peuvent donc être modélisées par des filtres organisés en pipeline. Pour l'initialisation de la somme, un premier filtre peut être utilisé, filtre un peu particulier qui à chaque étape envoie la même valeur d'initialisation (0). Les cellules doivent toutes recevoir des X_i, mais avec un décalage. Nous introduisons le filtre `BROADCAST` qui règle cet envoi. Tous les filtres communiquent via des pipes. La Figure 2 montre l'architecture ainsi conçue. Dans cet exemple, nous considérons juste trois valeurs ($n = 3$).

Suivant une approche descendante, nous avons un développement en quatre étapes :

- *Première étape* : identification de l'interface du système avec son environnement (instanciation de ***Ext_Gate_list***) ainsi que du type des valeurs échangées.
- *Deuxième étape* : identification des composants et des connecteurs et pour chacun d'eux définition de son interface, c'est à dire instanciation des caractéristiques des filtres et des pipes. A l'issue de cette étape, une variable ***beh*** représentant la configuration globale est définie.
- *Troisième étape* : définition du comportement de chaque filtre et pipe en choisissant un schéma de variation de filtre ou pipe.
- *Quatrième étape* : définition des types introduits.

À chaque étape, les contraintes données dans la définition du style doivent être satisfaites.

Première étape. Deux portes externes sont nécessaires : une porte d'entrée `X` pour recevoir la suite $(X_i)_i$ et une porte de sortie `Y` pour renvoyer la suite des résultats $(Yi)_j$. La liste des portes externes représentée par la variable ***Ext_Gate_list*** est égale à (`X, Y`). Sur `X` et sur `Y`, les valeurs échangées sont de type `EXP` décrivant des expressions symboliques.

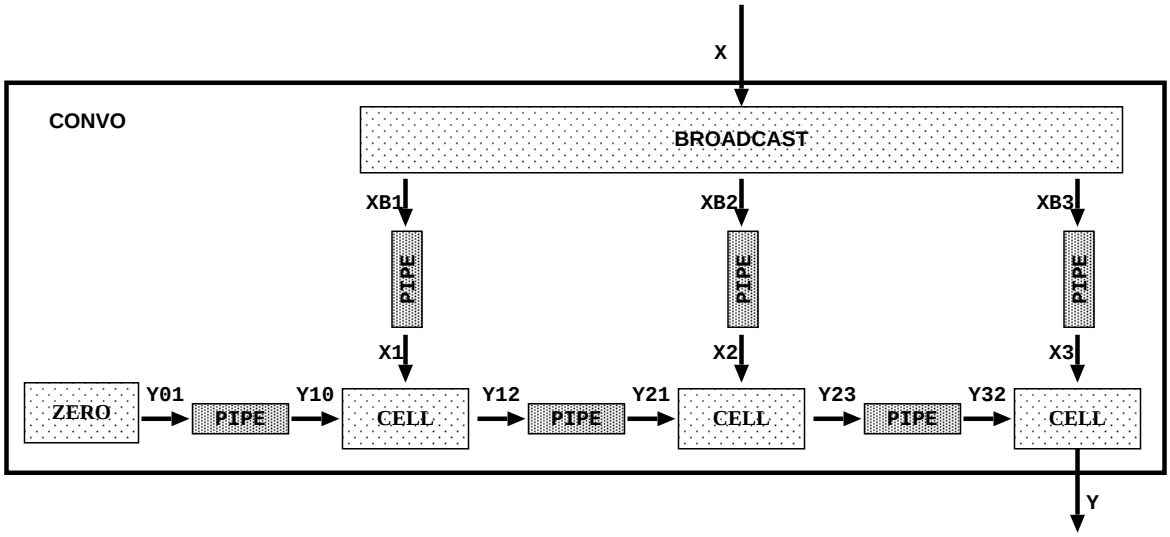

Fig. 2. Architecture de style *Pipe & Filter* pour le calcul d'un produit de convolution

Spécification du produit de convolution

```
specification Convo[X, Y] : noexit
  behaviour
     beh
  where
     local_def
endspec
```

Deuxième étape. Comme le montre la figure 2, trois types de filtres ont été identifiés: `BROADCAST`, `ZERO` et `CELL`. Pour chacun d'eux, les caractéristiques sont les suivantes:

- `BROADCAST` reçoit la suite $(X_i)_i$ par sa porte `X` et envoie ses valeurs aux cellules par l'intermédiaire des pipes sur les portes `XB1, XB2, XB3`. Les valeurs sont de type `EXP`. Le filtre `BROADCAST` dépend du nombre de cellules à qui il doit envoyer les valeurs.
- `ZERO` initialise le calcul du produit de convolution en envoyant la constante zéro à la première cellule par l'intermédiaire de la porte `Y01`.
- Chaque composante `CELL` calcule le produit d'une entrée avec son propre poids $\mathtt{W}_i$ et l'additionne avec le résultat courant reçu. Le nouveau résultat est envoyé à la cellule suivante. Ainsi, une cellule a deux entrées et une porte résultat. Chaque cellule est paramétrée par son poids $\mathtt{W}_i$.

Tous les filtres sont reliés par des pipes appelés `PIPE`. La table 1 résume l'instanciation des caractéristiques des composants et des connecteurs introduits.

Nous pouvons maintenant instancier la configuration globale donnée dans la section 3.2.

Composant	*gate_list_IN*: type	*gate_list_OUT*: type	type de l'état local
(Filter) BROADCAST	X : EXP	X1 : EXP, X2 : EXP, X3 : EXP	NAT
(Filter) ZERO	-	Y01 : EXP	-
(Filter) CELL	X1 : EXP, Y10 : EXP	Y12 : EXP	EXP(W_1)
(Filter) CELL	X2 : EXP, Y21 : EXP	Y23 : EXP	EXP(W_2)
(Filter) CELL	X3:EXP, Y32:EXP	Y:EXP	EXP(W_3)

Connecteur	*gate_IN*: : type	*gate_OUT*: : type
PIPE	Y01 : EXP	Y10 : EXP
PIPE	Y12 : EXP	Y21 : EXP
PIPE	Y23 : EXP	Y32 : EXP
PIPE	XB1 : EXP	X1 : EXP
PIPE	XB2 : EXP	X2 : EXP
PIPE	XB3 : EXP	X3 : EXP

Liste des portes externes : *Ext_Gate_list* = (X, Y)

Table 1. Caractéristiques du produit de convolution suivant le style *Pipe & Filter*

Configuration globale

```
beh =
      hide XB1, XB2, XB3, X1, X2, X3, Y01, Y10, Y12, Y21, Y23, Y32 in
       ((     BROADCAST [X, XB1, XB2, XB3](3)
          ||| ZERO [Y01]
          ||| CELL [X1, Y10, Y12] (W1)
          ||| CELL [X2, Y21, Y23] (W2)
          ||| CELL [X3, Y32, Y]   (W3) )
        |[ XB1, XB2, XB3, X1, X2, X3, Y01, Y10, Y12, Y21, Y23, Y32 ]|
        (     PIPE [XB1, X1]
          ||| PIPE [XB2, X2]
          ||| PIPE [XB3, X3]
          ||| PIPE [Y01, Y10]
          ||| PIPE [Y12, Y21]
          ||| PIPE [Y23, Y32] ))
```

local_def est défini comme étant la concaténation des définitions des processus BROADCAST, ZERO, CELL et PIPE ainsi que le définition du type EXP.

Notons que, avec cinq filtres ($p = 5$) et six pipes ($q = 6$), les contraintes données dans la section 3.2 sont satisfaites.

Troisième étape. Les spécifications sont obtenues de manière systématique par instanciation des schémas. Par manque de place, nous ne donnons pas toutes les définitions obtenues. La spécification complète peut être consultée dans le rapport interne [RL99].

Comportement des filtres.

- BROADCAST envoie aux cellules la valeur reçue. Il a un comportement particulier pendant l'initialisation[4]. Il peut être spécifié comme filtre bloquant ou non bloquant.

[4] Un certain nombre de retards égal au nombre de filtres est nécessaire, les i premières valeurs n'étant envoyées qu'aux i premières cellules :
$Y_1 = W_1 * X_1 + W_2 * X_2 + W_3 * X_3$
$Y_2 = W_1 * X_2 + W_2 * X_3 + W_3 * X_4$

– ZERO est un filtre dégénéré : il n'a aucune entrée, il envoie simplement une constante sur sa porte. Nous instancions le schéma général de filtre avec $n = 0$, $m = 1$ et $F = 0$. La définition du processus ZERO peut être simplifiée en enlevant le comportement de réception et celui de calcul.

```
def_zero =
          process ZERO [YO1] : noexit :=
              YO1 !(0 of EXP);
              ZERO [YO1]
          endproc
```

– Une cellule CELL additionne au résultat courant reçu le produit d'une entrée X_i avec son propre poids W_i et renvoie le résultat. Nous pouvons la spécifier comme filtre bloquant ou non bloquant. Son calcul est défini par la fonction F suivante :

F: EXP, EXP, EXP $\rightarrow$ EXP

$F\ (\ W_i,\ X_i,\ Y) = W_i\ *\ X_i\ +\ Y$

Comportement des pipes.
Nous avons six pipes. Chacune eux peut-être soit simple, soit borné, soit non borné. Une fois ce choix fait, l'instanciation est systématique. Notons que dans les cas de pipes avec des mémoires tampons, celles-ci doivent être initialisées à la valeur `init`, par exemple : une mémoire tampon non bornée est initialisée par `PIPE [XB1, X1](init of BUF)` et une mémoire tampon de 3 places par `PIPE [XB1, X1](init (3, init of BUF))`.

Quatrième étape. Définition des types introduits. Le type `EXP` décrit des expressions symboliques. Le type `NAT` est celui fournit par la bibliothèque LOTOS.

Annotations. Pour pouvoir vérifier a posteriori les contraintes architecturales du style, il est nécessaire d'annoter les processus définis par l'attribut *Archi*. Cet attribut est égal à `Pipe` pour le processus `PIPE` et à `Filter` pour les processus `BROADCAST`, `ZERO` et `CELL`. La spécification annotée satisfait toutes les contraintes architecturales données dans section 3.4. Ainsi, la spécification obtenue est conforme au style *Pipe & Filter*.

Analyse des spécifications définies.

Nous avons défini des spécifications en combinant les différents types de filtres et de pipes. Nous avons comparé les différentes solutions obtenues à l'aide de l'outil CADP (Caesar/Aldebaran Distribution Package) [FGM+92]. Toutes les solutions ont été démontrées équivalente relativement à la relation de sûreté (safety equivalence [FM91]). Le tableau 2 montre la taille des systèmes étiquetés de transitions (LTS) calculés par l'outil.

Les cas (a) et (b) diffèrent sur la définition du type de données `EXP`. En (a), la spécification inclut la définition du type comme un type abstrait de données tandis qu'en (b), celui-ci est défini directement en C. Les LTS sont équivalents.

$Y_3 = W_1 * X_3 + W_2 * X_4 + W_3 * X_5$
$Y_4 = W_1 * X_4 + W_2 * X_5 + W_3 * X_6$
...

Les cas (c) et (d) comparent les pipes bornés ou non. Dans (c) la taille de la mémoire tampon est de 3 places pour les pipes entre `BROADCAST` et les cellules et de 1 pour le pipe entre `ZERO` et la première cellule. Dans le cas (d), les pipes sont non bornés excepté pour celui relié à `ZERO` (car un pipe non borné autorise `ZERO` à envoyer infiniment ses zéros). Naturellement, dans le cas (d) le LTS est plus grand que celui du cas (c).

Les cas (e), (f) et (g) comportent des filtres non bloquants. Ceci augmente encore la taille des LTSs.

case	CELL	BROADCAST	type de données	PIPE	mémoire tampon	états	transitions
(a)	blocking	blocking	`EXP(ADT)`	simple	1	38694	160241
(b)	blocking	blocking	`EXP(C)`	simple	1	38694	160241
(c)	blocking	blocking	`EXP(C),BOUNDED_BUF(ADT)`	bounded	3	269311	1189813
(d)	blocking	blocking	`EXP(C),BUF(ADT)`	unbounded	-	331250	1466727
(e)	blocking	non-blocking	`EXP(C)`	simple	1	64188	285230
(f)	non-blocking	blocking	`EXP(C)`	simple	1	67756	315450
(g)	non-blocking	non-blocking	`EXP(C)`	simple	1	97798	469611

Table 2. Comparaison des variations de la spécifications

5 Conclusion

Nous avons présenté, à l'aide de l'exemple du style *Pipe & Filter*, une approche pour formaliser des styles architecturaux afin de guider la conception d'architectures de logiciels. Notre formalisation propose des schémas de spécifications paramétrés qui peuvent être instanciés pour développer une architecture. Plusieurs schémas de définition des composants ou des connecteurs, comportementalement équivalents, sont proposés, permettant ainsi au spécifieur de choisir entre plusieurs solutions. Nous appelons ces différents schémas des variations. Les caractéristiques du style, définies en tant que variables typées et contraintes, sont ses paramètres. Le fait de les rendre explicites, facilite l'instanciation du style. La spécification obtenue est par nature conforme au style choisi, c'est-à-dire que les contraintes architecturales sont satisfaites.

Les descriptions formelles des styles architecturaux sont importantes parce que c'est la seule manière de répondre avec précision aux questions posées par Clements [Cle96] : Quels sont les composants ? Comment se comportent-ils ? Quel type de connexions existe-t-il entre ces composants ?

Nous ne sommes pas les premiers à proposer des formalisations de styles d'architectures de logiciels. Des notations ad hoc ont été définies comme Wright [All97] ou Rapide [DJL+95]. Mais elles ne disposent pas vraiment d'environnements de développement comportant des outils d'analyse. Plusieurs notations formelles existantes ont également été utilisées. Entre autres, nous pouvons citer CSP [AG94], Z [DG91,AAG93], Larch [CP97], des grammaires de graphe [Met96] et même LOTOS [Tur98]. Un des avantages d'utiliser un formalisme existant, est la disponibilité d'environnements ou des outils de validation tels que CADP que nous avons utilisé. Dans tous les cas, ce qui semble important est de pouvoir spécifier aussi bien le traitement des données que la communication entre composants. LOTOS est un langage qui a réellement réussi à marier ces deux types de spécifications.

En ce qui concerne les variations, il n'y a pas beaucoup de travaux existants. La question est : quelles sont les relations entre les différentes variations? Nous proposons comme relation l'équivalence comportementale. Les schémas de raffinement définis dans [MQ94] pourraient aussi être considérés. Pour d'autres auteurs, les variations peuvent être aussi bien des relations définies lorsque l'on relâche une contrainte architecturale ou au contraire lorsque l'on en rajoute.

Une de nos préocupations majeures, est l'étude du processus de développement de spécifications. Dans l'exemple, nous avons esquissé une méthode descendante pour développer des architectures en quatre étapes. Cette approche pourrait être décrite par des *agendas* [Hei98] ou comme opérateurs de développement du modèle PROPLANE [SL96]. Dans PROPLANE, le développement d'une spécification est défini comme une succession d'étapes. Le passage d'un état de développement à un autre se fait par application d'un opérateur. Un opérateur formalise l'instanciation d'un schéma tels que ceux qui ont été introduits dans cet article. Dans l'environnement PROPLANE, le spécifieur choisit l'opérateur à appliquer parmi ceux d'une bibliothèque. Des contraintes sont associées aux opérateurs et vérifiées à chaque étape. Ceci rend le développement aisé et sûr.

References

[AAG93] G. Abowd, R. Allen, and D. Garlan. Using style to understand descriptions of software architecture. In *Proceedings of SIGSOFT'93: Foundations of Software Engineering*, Software Engineering Notes 18(5), pages 9–20. ACM Press, December 1993.

[AG94] R. Allen and D. Garlan. Formalizing architectural connection. In *Proceedings 16th Int. Conf. on Software Engineering*. ACM Press, 1994.

[All97] Robert J. Allen. *A Formal Approach to Software Architecture*. PhD thesis, Carnegie Mellon University, May 1997.

[BB87] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14:25–59, 1987.

[Cle96] P. Clements. A survey of architecture description languages. In *Proceedings of the 8th International Workshop on Software Specification and Design*, pages 16–25, Schloss Velen, Germany, March 1996. IEEE.

[CP97] P. Ciancarini and W. Penzo. Validating a software architecture with respect to an architectural style. Technical Report UBLCS-97-7, University of Bologna (Italy). Department of Computer Science., URL:ftp://ftp.cs.unibo.it/pub/techreports/97-07.ps.gz, July 1997.

[DG91] D. Notkin D. Garlan. Formalizing design space: Implicit invocation mechanisms. In W. Toetenel S. Prehn, editor, *Proceedings of the 4th Annual Symposium: VDM '91*, volume 551 of *Lecture Notes in Computer Science*, pages 31–44. sv, October 1991. 1991.

[DJL+95] D. C. Luckham, J. J. Kenny, L. M. Augustin, J. Vera, D. Bryan, and W. Mann. Specification and analysis of system architecture using rapide. *IEEE Transactions on Software Engineering*, pages 336–355, April 1995.

[EGY97] A. Eden, J. Gil, and A. Yehudai. Automating application of design patterns. *Object-Oriented Programming*, 10(2), May 1997.

[EM85] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1 (Equations and Initial Semantics)*, volume 6 of *ETACS Monographs on Theorical Computer Science*. Springer Verlag, 1985.

[FGM+92] J-C. Fernandez, H. Garavel, L. Mounier, A. Rasse, C. Rodríguez, and J. Sifakis. A toolbox for the verification of lotos programs. In Lori A. Clarke, editor, *Proceedings of the 14th International Conference on Software Engineering ICSE'14 (Melbourne, Australia)*, pages 246–259. ACM, May 1992.

[FM91] J.-C. Fernandez and L. Mounier. A tool set for deciding behavioral equivalences. In *Proceedings of CONCUR'91 (Amsterdam, The Netherlands)*, August 1991.

[Gar89] H. Garavel. *Compilation et vérification de programmes LOTOS*. Thèse de doctorat, Université Joseph Fourier (Grenoble), November 1989.

[GS93] D. Garlan and M. Shaw. An introduction to software architecture. *Advances in Software Engineering and Knowledge Engineering, World Scientific Publishing Company*, 1, 1993.

[Hei98] M. Heisel. Agendas – a concept to guide software development activites. In R. N. Horspool, editor, *Proc. Systems Implementation 2000*, pages 19–32, London, 1998. Chapman & Hall.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes.* Series in Computer Science. Prentice-Hall International, 1985.

[Kun82] H. T. Kung. Why systolic architectures ? *Computer*, 15(1):37–46, january 1982.

[LOT87] ISO. LOTOS. A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. Draft international standard 8807, International Organization for Standardization - Information Processing Systems - Open Systems Interconnection, Geneva, 1987.

[Me98] J. Magee and D. Perry editors, editors. *Third International Software Architecture Workshop (ISAW 3)*, Orlando, Florida, USA, Nov 1998. SIGSOFT, ACM Press.

[Met96] D. Le Metayer. Software architecture styles as graph grammars. In *In Proc of the ACM SIGSOFT Symposium of the foundations of Software Engineering*, pages 15–23, 1996.

[Mil80] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science.* Springer-Verlag, 1980.

[MQ94] M. Moriconi and X. Qian. Correctness and composition of software architectures. In David Wile, editor, *Proc. of the Second ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE'94)*, pages 164–174. ACM Press, 1994.

[NR97] N. Medvidovic and R. Taylor. A framework for classifying and comparing architecture description languages. In *ACM Sigsoft Software Engineering Notes. ESEC/FSE'97*, volume 22 no 6, November 1997.

[RL99] Rongviriyapanish Songsakdi and Lévy Nicole. Variations on the pipe and filter architectural style. Technical Report 99-R-027, LORIA, Campus Scientifique, B.P. 239, 54506 Vandoeuvre-les-Nancy, France, 1999.

[SG96] M. Shaw and D. Garlan. *Software Architecture, perspectives on an emerging discipline.* Prentice Hall, 1996.

[SL96] J. Souquières and N. Lévy. PROPLANE : A Specification Development Environment. In *Fifth Int. Conf. on Algebraic Methodology and Software Methodology (AMAST'96)*, volume 1101, Munich (G), July 1996. Lecture Notes in Computer Science.

[Tur98] K. J. Turner. Validating architectural feature descriptions using LOTOS. In Kristofer Kimbler and Wiet Bouma, editors, *Proc. 5th. International Workshop on Feature Interactions in Telecommunication Networks and Software Systems*, pages 247–261. IOS Press, Amsterdam, Netherlands, September 1998.

Résumé du langage LOTOS utilisé dans cet article

Une spécification LOTOS décrit le comportement de processus inter-agissants les uns avec les autres. Un processus peut être paramétré par des types abstraits de données, et il peut échanger des valeurs typées avec d'autres processus. Il peut aussi appeler des fonctions pour transformer des données. La communication entre les processus est synchrone, c'est-à-dire que deux processus doivent participer à une action commune en même temps. Des portes sont utilisées pour synchroniser les processus et pour l'échange de données. Chaque définition de processus a la forme syntaxique suivante :

```
process process_name[gate_list](params): func :=
  behaviour
    beh
  where
    local_def
endproc
```

où ***func*** indique si le processus se termine (***func*** = `exit` ou `exit(v)` s'il se termine en envoyant une valeur `v`) ou non (***func*** = `noexit`).

L'expression de comportement ***beh*** décrit la suite des actions observables qui peuvent se produire. Un comportement peut être une instanciation d'un processus, une action de communication entre comportements ou être la composition de plusieurs comportements reliés par des opérateurs.

L'opérateur de choix `[]` est utilisé quand des comportements alternatifs sont autorisés. L'expression `P1 [] P2` signifie qu'exactement un des deux comportements `P1` ou `P2` sera exécuté, selon un choix de l'environnement.

L'expression d'entrelacement `P1 ||| P2` (interleaving) exprime que les deux comportements `P1` et `P2` se comportent indépendamment et en parallèle.

L'expression de synchronisation `P1 |[G]| P2` (parallel composition) exprime que les deux comportements `P1` et `P2` doivent se synchroniser sur la porte `G`. Pendant la synchronisation, ils peuvent échanger des données. Pour se synchroniser, les deux comportements doivent contenir une action sur la même porte `G`. Pour échanger des données, l'un d'entre eux doit contenir une action `G ? v: T` qui lit une valeur `v` du type `T` sur la porte `G` . L'autre comportement doit contenir une action `G ! exp` qui envoit la valeur de l'expression `exp` qui doit également être de type `T` sur la porte `G` . Il est possible de lire ou d'écrire plusieurs valeurs dans une même action, par exemple en écrivant `G ? v: T ? w: T'`. Une action peut être gardée par un prédicat *pred* : `G ? v: T [`*pred*`]`.

Des comportements peuvent être rendus conditionnels en utilisant l'opérateur de garde `[`*pred*`] ->` *beh*. Dans ce cas, l'expression *beh* aura lieu seulement si le prédicat *pred* est vérifié.

Deux comportements peuvent être composés séquentiellement par l'opérateur ≫. L'expression `P1` ≫ `P2` exprime le fait que `P1` doive se terminer pour que `P2` soit exécuté. L'expression `P1` ≫ `accept x: T in ( P2 )` ou `P1` se termine par `exit (v)` décrit la transmission de la valeur de la variable `v` du comportement `P1` au comportement `P2` pour lequel cette valeur est celle de la variable `x`.